Rescuing LoRaWAN 1.0

Gildas Avoine[1,2,3] and Loïc Ferreira[4,2]

[1] INSA Rennes, CNRS, France
[2] IRISA UMR 6074, Rennes, France
[3] Institut Universitaire de France
[4] Orange Labs, Applied Cryptography Group, Caen, France

Abstract. LoRaWAN is a worldwide deployed IoT security protocol. We provide an extensive analysis of the version 1.0, which is the currently deployed version, and we show that it suffers from several weaknesses. We introduce several attacks, including practical ones, that breach the network availability, data integrity, and data confidentiality, and target either an end-device or the backend system.
Based on the inner weaknesses of the protocol, these attacks do not lean on potential implementation or hardware bugs. Likewise they do not entail a physical access to the targeted equipment and are independent from the means used to physically protect secret parameters.
Finally we propose practical recommendations aiming at thwarting the attacks, while at the same time being compliant with the specification, and keeping the interoperability between patched and unmodified equipment.

1 Introduction

1.1 Context

With the arrival of the Internet of Things, several communication protocols have been proposed, with technical specifics suited to the intended use case. For instance, the Bluetooth wireless protocol [3] is designed for short distance communication. Technologies such as ZigBee [30] or Z-Wave [29] afford medium range distance communication, and aim at reducing the energy needed by the nodes to set up and maintain a mesh network.

As for long range distance communication (several kilometers), proposals have been made, such as LoRa. LoRa, developed by Semtech company, aims at setting up a Low-Power Wide-Area Network (LPWAN) based on a long-range, low-rate, wireless technology. It is somewhat similar to a cellular technology (2G/3G/4G mobile systems) but optimised for IoT/M2M. LoRa does not require a spectrum license because it uses free (although regulated) frequency bands (e.g., 863-870 MHz in Europe, 902-928 MHz in the USA, 779-787 MHz in China) [14]. A LoRa end-device, with an autonomous power-supply, is supposed to communicate through several kilometers in an urban area, and to have a lifespan up to eight or ten years.

LoRaWAN is a protocol that aims at securing the Medium Access Control

layer of a LoRa network. It is designed by the LoRa Alliance, which is an association that gathers more than 400 members (telecom operators, semiconductor manufacturers, digital security companies, hardware manufacturers, network suppliers, etc.).

Public and private LoRaWAN networks are deployed in more than 50 countries worldwide [23] by telecom operators (SK Telecom, FastNet, ZTE, KPN, Orange, Proximus, etc.), private providers (e.g., LORIOT.io [15]), and private initiatives (e.g., The Things Network [27]). Several nationwide networks are already deployed in Europe (France, Netherlands) [7], Asia (South Korea) [16], Africa (South Africa) [1], Oceania (New Zealand) [24], providing coverage to at least half of the population. Trials are launched in Japan [4], the USA (starting with a hundred cities) [10], China (the expected coverage extend to 100 million homes and 300 million people) [22], India (the first phase network aims at covering 400 million people across the country) [9].

The services provided by LoRa end-devices are numerous, from performing measurements (humidity, temperature, water leak, etc.), up to achieving more sensitive purposes such as triggering an alarm/help message, detecting an intrusion, or allowing to remotely switch on and off another equipment. The data sent by the end-device may also have to remain confidential (e.g., geolocation of valuable assets sent by a tracker).

In this paper we focus on the version 1.0.2 of the LoRaWAN specification released in 2016, which is the version currently deployed worldwide, and whose official name is now 1.0.

1.2 Protocol overview

A LoRaWAN network corresponds to a star-of-stars topology: a set of end-devices communicates with several gateways, which relay the data to a Network Server (NS) in the backend. In turn, the NS delivers the data to one or more Application Servers (AS), which own the corresponding end-devices, optionally through intermediary servers such an MQTT server (see Figure 1). The security mechanisms are based on a symmetric key `AppKey` (the root key) shared between an end-device and the NS. From this key, distinct per end-device, two session keys are computed: the application session key `AppSKey` guarantees the data confidentiality between the end-device and the AS; the network session key `NwkSKey` guarantees the data integrity between the end-device and the NS (it is worth noting that data integrity is not provided end-to-end between the end-device and the AS[5]). When a frame is exchanged exclusively between an end-device and the NS, both data confidentiality and data integrity are provided by the network session key `NwkSKey`. An application payload is always encrypted. Moreover, if no payload is carried, the frame is only authenticated. Encryption is done with AES [19] in CTR mode [5, 20], and data integrity is provided with AES in CMAC mode [21, 25]. An end-device may establish an "activation" (namely a session) with the NS through two ways. The pre-personalization (Activation By Personalization,

[5] As acknowledged by the specification ([26], §6.1.4).

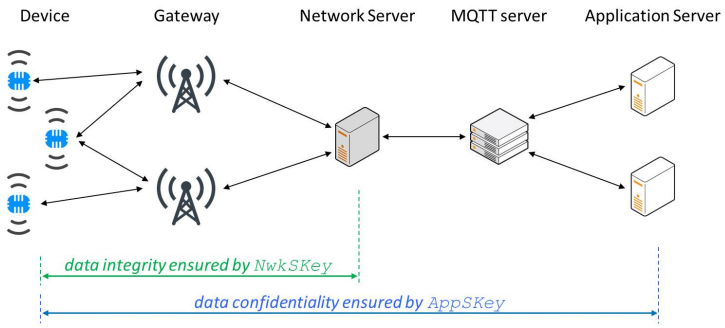

Fig. 1. LoRaWAN network (simplified view)

ABP) consists in setting two session keys (and other parameters but not the `AppKey` root key) into the end-device before its deployment. An ABP end-device is then able to communicate with the NS (and its AS) but not to renew the "session" keys. The other possibility (Over The Air Activation, OTAA) consists in provisioning the end-device with an `AppKey` root key and other parameters, allowing for key exchanges with the NS through the radio interface once it is deployed. In this paper we focus on OTAA end-devices.

1.3 Paper outline

The LoRaWAN protocol is detailed in Sect. 2. Theoretical and practical attacks against LoRaWAN are described in Sect. 3. Sect. 4 describe recommendations that thwart the attacks. Sect. 5 summarises previous comments and analysis on the protocol. Sect. 6 deals with the responsible disclosure. We finally conclude in Sect. 7.

2 The LoRaWAN protocol

The technical features described in this section are based on [26].

2.1 Key exchange

The key exchange done over the air is triggered when the end-device sends a Join Request message. The NS then responds with a Join Accept message. The (unencrypted) Join Request message includes two static identifiers (the end-device's `DevEUI` and the AS' `AppEUI`), and a pseudo-random value `DevNonce` generated by the end-device. The message is protected with a 4-byte CMAC authentication tag (called `MIC`) computed with the 128-bit (static) root key `AppKey`. The Join Accept response from the NS contains the (static) identifier of the latter (`NetID`), a pseudo-random value generated by the NS (`AppNonce`), a value used

as the end-device short address (DevAddr), and several (optional) radio parameters. The Join Accept message is protected with a CMAC authentication tag, and encrypted with AES (both operations made with the root key AppKey).[6] Two 128-bit session keys are then computed:

$$\texttt{NwkSKey} = \mathsf{AES}(\texttt{AppKey}, \texttt{0x01}\|\texttt{data})$$
$$\texttt{AppSKey} = \mathsf{AES}(\texttt{AppKey}, \texttt{0x02}\|\texttt{data})$$

with $\texttt{data} = \texttt{AppNonce}\ (3)\|\texttt{NetID}\ (3)\|\texttt{DevNonce}\ (2)\|\texttt{0x00}\cdots\texttt{00}\ (7)$.

Thus the session keys depend mostly on a secret and static value (the root

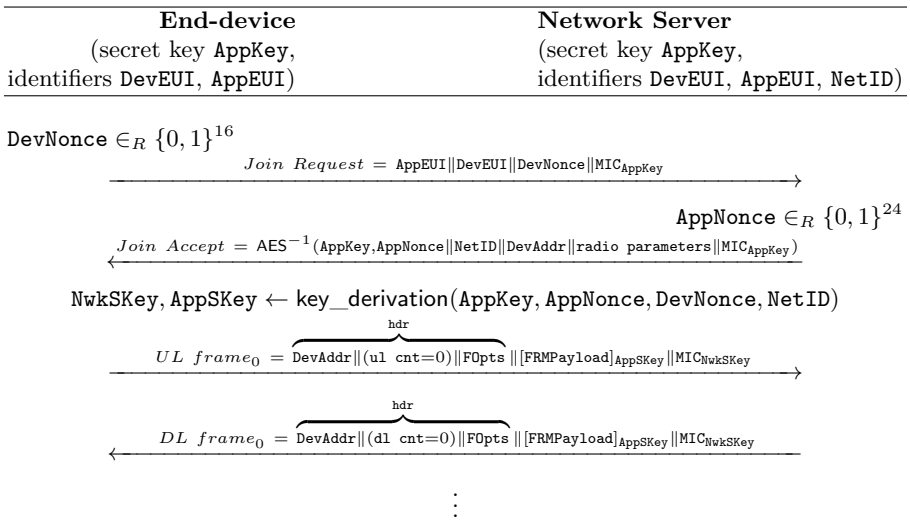

Fig. 2. LoRaWAN activation (simplified scheme).

key AppKey), and two pseudo-random values, respectively 2-byte and 3-byte long. Once the Join Request and Join Accept messages are exchanged, the end-device, the NS, and the AS are able to communicate. After the NS computes the session keys, it transmits the application session key AppSKey to the AS. It is worth noting that the AS is not involved in the key agreement, which is handled by the NS. The NS must keep the previous session keys, and the corresponding security parameters, until it receives a (valid) frame protected by the new security parameters. The security mechanisms between NS and AS are out of the LoRaWAN scope. Figure 2 depicts an activation.

[6] More precisely the AES decryption function is used to protect the Join Accept message, because the end-device implements the encryption function only.

2.2 Data encryption and authentication

The frame payload FRMPayload is encrypted in CTR mode. From block counters

$$A_i = \texttt{0x01}\ (1)\|\texttt{0x00}\cdots\texttt{00}\ (4)\|\texttt{dir}\ (1)\|\texttt{DevAddr}\ (4)\|\texttt{cnt}\ (4)\|\texttt{0x00}\ (1)\|i\ (1)$$

a secret keystream $S_i = \mathsf{AES}(K, A_i)$ is produced with $K \in \{\texttt{AppSKey}, \texttt{NwkSKey}\}$, and used to mask the payload[7]: $[\texttt{FRMPayload}] = (S_0\| \cdots \|S_{n-1}) \oplus \texttt{FRMPayload}$.

dir specifies the direction (uplink = 0x00, downlink = 0x01). cnt is the frame counter (of 16 or 32 bits), initialised to 0 when the session starts, and monotonically increased when a (valid) frame is sent or received. Two different counters are used depending on the frame's direction. DevAddr is the end-device address (within a given LoRa network) chosen by the NS and sent in the Join Accept message, and it remains constant during the entire session. To compute DevAddr, seven bits are chosen from the NS' unique identifier NetID: $\mathsf{msb}_7(\texttt{DevAddr}) = \mathsf{lsb}_7(\texttt{NetID})$, and 25 bits are "*arbitrarily*" assigned by the NS. The i value numbers the AES blocks within the payload to encrypt.

A 4-byte authentication tag is computed with CMAC and the network ses-

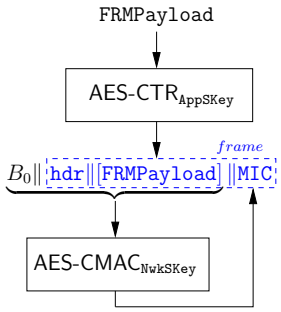

Fig. 3. Generation of an application frame.

sion key NwkSKey on the whole frame (header hdr of size $hlen \in \{8, \ldots, 24\}$ and encrypted payload [FRMPayload] of size $plen$) and a 16-byte prefix block

$$B_0 = \texttt{0x49}\ (1)\|\texttt{0x00}\cdots\texttt{00}\ (4)\|\texttt{dir}\ (1)\|\texttt{DevAddr}\ (4)\|\texttt{cnt}\ (4)\|\texttt{0x00}\ (1) \\ \|(hlen + plen)\ (1)$$

The frame eventually sent is hdr $(hlen)\|[\texttt{FRMPayload}]\ (plen)\|\texttt{MIC}\ (4)$.

The frame header hdr includes, among other fields, DevAddr, the frame counter cnt on 2 bytes, and an (optional) field FOpts which may contain commands exclusively exchanged between the end-device and the NS (see Figure 3).

[7] The session key K = AppSKey is used when application messages are exchanged between the end-device and the AS, and K = NwkSKey is used when command-only messages are exchanged between the end-device and the NS.

3 Attacks against LoRaWAN

Hereinafter we present our findings regarding the LoRaWAN protocol version 1.0, the currently deployed version. Table 1 summarises the attacks we have found.

Our attacker, standing between a LoRaWAN end-device and the NS, needs only to act on the air interface: she needs to eavesdrop on data exchanged between the end-device and the server, and to send data to any equipment.

Table 1. Attacks against LoRaWAN 1.0 (n_{ja}: number of Join Accept messages usable by the attacker. n_{jr}: number of Join Request messages usable by the attacker. m: number of new session keys sets stored by the NS. ED: end-device)

Attack	Complexity (# Join message)	Probability of success	Impact
(A1) Replay or decrypt (ED, Sect. 3.1)	$2^{16}/n_{ja}$	$\simeq 1$	downlink frame replay, uplink frame decryption
(A2) Replay or decrypt (NS, Sect. 3.1)	n_{jr}	$\simeq n_{jr}/2^{24}$	uplink frame replay, downlink frame decryption
(A3) Desynchronization (ED, Sect. 3.2)	1	1	end-device desynchronization
(A4) Desynchronization (NS, Sect. 3.2)	m	1	end-device desynchronization

3.1 Replay or decrypt

Targeting an end-device (attack A1)

Goal The purpose of this attack is to compel the end-device to reuse previous session keys and other security parameters. When this happens, frames picked from a previous session become cryptographically valid anew, hence can be replayed. Moreover the same secret keystream is then used to protect the frames exchanged during the new session. This allows an adversary to decrypt frames.[8]

Core The encryption keystream $S_i = \mathsf{AES}(K, A_i)$ used to protect a frame payload is produced from a session key $K \in \{\texttt{AppSKey}, \texttt{NwkSKey}\}$ and A_i block counters. Within a given session the blocks

$$A_i = \texttt{0x01}\ (1) \| \texttt{0x00} \cdots \texttt{00}\ (4) \| \texttt{dir}\ (1) \| \texttt{DevAddr}\ (4) \| \texttt{cnt}\ (4) \| \texttt{0x00}\ (1) \| i\ (1)$$

[8] Note that since the end-device ends up reusing previous session keys (which are no longer shared with the NS), this attack is also a kind of "desynchronization" attack. However, contrary to the desynchronization attacks described in Sect. 3.2, this "replay or decrypt" attack has more devastating consequences (and a higher complexity) than "only" desynchronizing the end-device and the NS.

(as well as the prefix block B_0) depend mostly on the frame counter `cnt` (set to 0 when the session starts and monotonically increased frame after frame), and on the `DevAddr` parameter (static during the whole session). The other parameters are the direction `dir` unchanged for a given direction, and the i block index which evolves the same way for each frame. Hence the way the keystream S_i changes depends only on the `DevAddr` parameter and the session key (usually `AppSKey`). For a given end-device, which connects to the same NS (hence uses the same static `NetID` parameter), the session keys depend mainly on a secret and static value (`AppKey`) and two pseudo-random values (`DevNonce`, `AppNonce`). Therefore, if one succeeds in compelling the end-device to reuse the same `DevAddr`, `DevNonce` and `AppNonce` parameters, this leads not only to the reuse of previous session keys `AppSKey`, and `NwkSKey`, but also to the reuse of previous keystream S_i and prefix block B_0.

Attack The purpose is to make the end-device use twice the same `DevNonce`, `AppNonce`, and `DevAddr` values. The 2-byte `DevNonce` and 3-byte `AppNonce` parameters are pseudo-random. Let us assume that an attacker is able to impose the `AppNonce` value that the end-device uses to compute the session keys. The probability that the session keys repeat depends then only on the `DevNonce` parameter. Firstly note that a collision due to the birthday paradox happens with high probability ($p = \frac{1}{2}$) after roughly $\sqrt{2\ln(2) \times 2^{16}} \simeq 301$ activations only. However the attacker can speed up the whole process: the attacker eavesdrops on a given session, and compels the end-device to generate multiple `DevNonce` values until the expected value is produced once again. In such a case only one value among 2^{16} is useful to the attacker. Hence, the end-device must generate on average 2^{16} `DevNonce` values. If the attacker eavesdrops on n_{ja} different Join Accept messages (each one corresponding to a different `DevNonce` value) previously received by the targeted end-device, then the probability for the attacker to succeed is $p = n_{ja}/2^{16}$. If the attacker repeats this experiment k times, the probability to be successful at least once among the k experiments is $1 - (1-p)^k \simeq kp$ if p is close to 0. In order this probability to be close to 1, the number of experiments the attacker has to perform is $k \simeq 2^{16}/n_{ja}$. This means that the end-device has to send $2^{16}/n_{ja}$ Join Request messages before one carries a `DevNonce` value that matches with one of the Join Accept messages.

The shortest receiving window of a Join Accept message is 5 seconds [14]. If the attacker uses $n_{ja} = 16$ Join Accept messages, the attack is achieved after roughly $2^{16}/16 \times 5$ seconds = 5.7 hours (assuming that the time needed to process the Join messages is negligible compared to the communication duration).[9]

Once this first phase of the attack is achieved, the attacker ends with two different sessions protected with the same security parameters, denoted respectively s_{old} and s_{new}.

Technique 1 used to achieve the attack: replay of a Join Accept message In order to compel the end-device to use a given `AppNonce` value, the attacker can replay

[9] See Appendix A.

a previous Join Accept message sent to the targeted end-device. Then the end-device will reuse (once again) the parameters included in the message. Indeed the data carried in a Join Accept message correspond to

$$\texttt{AppNonce}\ (3)\|\texttt{NetID}\ (3)\|\texttt{DevAddr}\ (4)\|\texttt{radio parameters}\ (2\ldots18)\|\texttt{MIC}\ (4)$$

where `MIC` is an authentication tag computed on the preceding fields with the (static) root key `AppKey`. These parameters are protected with AES and `AppKey`. The cornerstone of this attack is that all the parameters are chosen by the NS, in particular `AppNonce` and `DevAddr`. `NetID` is the NS' (static) identifier, and the `radio parameters` are also defined by the NS. The only secret parameter involved in the message calculation is static (`AppKey`). Hence, the end-device is not able to verify if the received Join Accept message corresponds to the Join Request it sent. Replaying a Join Accept message allows the attacker to compel the end-device to (re)use both `AppNonce` and `DevAddr` parameters.

Technique 2 used to achieve the attack: harvest of Join messages The ability of the attacker to make the end-device generate multiple `DevNonce` values is related to the behaviour of the end-device when it sends a Join Request message but does not receive a Join Accept response or receives an invalid message. The specification states that the NS shall ignore Join Request messages containing previously used `DevNonce` values in order to thwart a replay attack ([26], §6.2.4). Hence, the end-device has to generate a new pseudo-random `DevNonce` value each time it computes a Join Request message, even when a previous Join Request message did not receive a response. Otherwise the end-device may fear the subsequent Join Request messages to be dropped by the NS. This allows the attacker to collect multiple new and valid Join Request messages. It is enough for the attacker to send "false" (i.e., invalid) Join Accept messages in response to the end-device's messages. Moreover, if the attacker forbids the NS from receiving the Join Request messages sent by the end-device, he gets "fresh" messages (i.e., unknown to the NS) for free. In order to make the end-device start producing the Join Request messages, the attacker may wait or force (once only) the end-device to start a new session (e.g., the attacker may turn the end-device off and on: once the power supply is re-established, the end-device likely starts a new activation).[10]

Note that every time the NS receives a Join Request message, it sends a new Join Accept message. Therefore, this procedure is also a way to collect multiples Join Accept messages.

[10] Being able to influence on the power supply does not necessarily mean to physically intrude on the end-device. The attacker could turn off or interrupt a remote electric generator the end-device is connected to, or the link between the generator and the end-device (if the end-device is powered by an external source), or use other means (e.g., electromagnetic impulse targeting the end-device and leading to a power outage).

Impact: frame replay Frames drawn from the previous session (s_{old}) can be replayed to the end-device throughout the new session (s_{new}).[11] These frames are valid since they are protected with a cryptographically correct keystream and authentication tag.

Impact: frame decryption The frame payload is encrypted in CTR mode. Once the attack is achieved, the end-device uses twice the same keystream in order to protect different frames. The frame of counter t sent during session s_{old} contains an encrypted payload $c_t^{s_{old}} = m \oplus k_t^{s_{old}}$, where m is the clear data and $k_t^{s_{old}}$ the keystream. The frame of same counter t sent during session s_{new} contains an encrypted payload $c_t^{s_{new}} = m' \oplus k_t^{s_{new}}$. Since $k_t^{s_{old}} = k_t^{s_{new}}$, we have that $c_t^{s_{old}} \oplus c_t^{s_{new}} = (m \oplus k_t^{s_{old}}) \oplus (m' \oplus k_t^{s_{new}}) = m \oplus m'$. Therefore m and m' may (partially or completely) be retrieved, in an obvious manner if either message is known, or through analysis of $m \oplus m'$ [17].

Targeting the NS (attack A2)

Goal The same kind of attack can be performed against the NS, aiming at compelling the server to use the same security parameters throughout two different sessions. The goal is then to compel the NS to use twice the same `DevNonce`, `AppNonce`, and `DevAddr` values.

Attack An attacker who replays a Join Request message sets the `DevNonce` value before knowing the `DevAddr` and `AppNonce` values generated by the NS. These values must correspond to the `DevNonce` value chosen by the attacker. Hence only one such pair among all possible values is of interest to the attacker.

According to the specification, the NS must keep track of "*a certain number*" of received `DevNonce` values in order to prevent replay attacks, without clarifying if this means all values or a few of them. We may reasonably assume that the NS keeps track of a few values (say n). Thus the attacker cannot choose any Join Request she wants to replay. The corresponding `DevNonce` value must not belong to the list of n stored values. If the value the attacker wants to replay still belongs to the server's list (let i be its index, with 0 and $n-1$ the index of the oldest and of the latest received values), she has to wait for $i+1$ additional (legitimate) key exchanges before the NS "forgets" that value. The duration of such an "opportunist" attack depends on the frequency of the key exchanges.

According to the specification, `AppNonce` is a 3-byte pseudo-random value, and the 32-bit `DevAddr` parameter is made of 7 bits from `NetID`, and 25 bits "*arbitrarily*" chosen by the NS ([26], §6.1.1). If `DevAddr` is pseudo-random then the probability of success is $2^{-(24+25)} = 2^{-49}$. But "*arbitrarily*" does not mean "pseudo-random" and experiments we have performed show that the `DevAddr`

[11] We use the term "session" for the sake of simplicity, but it does not depict precisely what are the actual exchanges since the end-device, at this point, has no "partner": neither the NS nor the AS is able to communicate with the end-device, and the attacker is unable to forge new valid frames.

parameter may remain unchanged for a given end-device throughout different sessions.[12] In such a case the probability of success increases to 2^{-24}, and the overall probability of success is 2^{-24} every $n+1$ sessions. Alternatively, the attacker can eavesdrop on n_{jr} different Join Request messages (that the NS has "forgotten"), and send them to the server.[13] The probability that at least one message triggers the same `AppNonce` value as during a previous session is $1-(1-2^{-24})^{n_{jr}} \simeq n_{jr} \times 2^{-24}$. For instance, if the attacker uses $n_{jr} = 2048$ Join Request messages, her probability to succeed raises to $\frac{1}{8192}$.

This attacker knows if the `AppNonce` value repeats through the direct comparison of the Join Accept messages, even if these messages are encrypted. Indeed, all the parameters of such a message are likely static but the `AppNonce` parameter.

Impact Once the attacker succeeds in compelling the NS to compute once again the same security parameters, she eventually gets two different sessions (s_{old} and s_{new}) protected with the same security parameters. The attacker is then able to replay uplink frames and attempt decryption of downlink frames. Note that the attacker is then able to send (i.e., to replay) a valid frame that indicates the NS to switch from the current security context to the new one (hence the NS drops the current session keys and uses the new ones).

3.2 Desynchronization

Targeting an end-device (attack A3)

Goal This attack aims at "disconnecting" the end-device from the network. That is the end-device performs a successful key exchange which ends with the end-device not sharing the new session keys with the NS (the end-device has no "partner"). Therefore the frames sent by the end-device are ignored by the NS, and conversely.

Core The session keys are computed, by a given end-device and the NS, with two static parameters (the NS' unique identifier `NetID`, and the end-device's root key `AppKey`), and two variable parameters (the pseudo-random values `AppNonce` computed by the NS, and `DevNonce` by the end-device). As soon as the end-device receives a (valid) Join Accept message it can derive the session keys and start transmitting protected frames. In the key derivation, if the end-device uses values different from those actually sent by the NS (say $(\texttt{DevNonce}, \texttt{AppNonce}) = (x, \tilde{y})$ on the one hand, and $(\texttt{DevNonce}, \texttt{AppNonce}) = (x, y)$, on the other hand, $y \neq \tilde{y}$), it eventually computes different session keys than those computed by

[12] Thus some NS implementation derives the `DevAddr` parameter from the unique end-device's identifier `DevEUI`. Also the `DevAddr` value may be chosen once and for all at the time of the end-device provisioning.

[13] The messages may come from different end-devices, hence, may have to be sent to one or several NS servers.

the server. This does not forbid the end-device to send protected frames though. However those frames will be dropped by the NS since they are invalid from the server perspective. Conversely, the frames sent by the NS will be discarded by the end-device. Thus the end-device, unable to communicate with the NS, is "disconnected" from the network.

Attack In order to perform such a desynchronization attack, an attacker can first passively eavesdrop on a Join Accept message sent by the NS in response to the end-device's Join Request message. When the end-device starts a new session and sends another Join Request message, the attacker replies before the NS and replays the eavesdropped Join Accept message. This replayed message likely contains an $\texttt{AppNonce} = \tilde{y}$ value different from the fresh one sent by the NS ($\texttt{AppNonce} = y$). Hence, the end-device and the NS compute different session keys and security parameters.

Means used to achieve the attack The attacker is able to replay a previous Join Accept message thanks to the peculiarities of the LoRaWAN protocol: indeed the end-device has no means to verify neither if the message is a replay, nor if it is an actual response to the Join Request message it just sent. Moreover the attacker can use the procedure described in Sect. 3.1 to collect several Join Accept messages and use these "desynchronization ammunition" anytime later. The Join Accept message used by the attacker must be intended to the targeted end-device. Indeed such a message is protected with the root key of the end-device it is sent to.

Impact Such a desynchronization attack may be harmful because it can lastingly disturb the operating of a LoRaWAN network. So then the usual behaviour of a sensor may be to regularly send some measurements without expecting a response unless the server detects an anomaly in the collected data. If the end-device sends its measurement at a low rate, days or even weeks may elapse before something abnormal is noticed, even if the end-device is supposed to react if it does not receive a downlink frame after a fixed number of sent frames.

Targeting the NS (attack A4)

Goal The same kind of desynchronization attack can be done against the NS, aiming at disconnecting a given end-device from the network. In that case, the NS completes the key exchange without being "partnered" with the intended end-device (i.e., identified by the `DevEUI` parameter within the Join Request message). Therefore the frames the NS (or the AS) may send are ignored by the end-device, and conversely.

Attack Upon reception of a (valid) Join Request message, the NS generates a new `AppNonce` value and computes new session keys. If an attacker succeeds in replaying to the NS a valid Join Request message, the corresponding end-device

will no longer share the same session keys with the NS. The attacker can do the following. She waits for the end-device to start a new activation. New session keys ($\texttt{seskeys}_{i+1}$) are then computed. The end-device stores $\texttt{seskeys}_{i+1}$ only while the NS stores both $\texttt{seskeys}_i$ and $\texttt{seskeys}_{i+1}$ (respectively the current and the new session keys). Before the end-device sends a frame, the attacker immediately sends to the NS a Join Request message she previously eavesdropped on (and not received, hence new to the server). The server computes new session keys $\texttt{seskeys}_{i+2}$ which replace the unconfirmed keys $\texttt{seskeys}_{i+1}$. Then the NS stores $\texttt{seskeys}_i$ and $\texttt{seskeys}_{i+2}$ while the end-device stores $\texttt{seskeys}_{i+1}$. Hence the end-device and the NS do not share the same session keys. More generally, if the NS keeps the latest valid session keys and m new sets of keys, the attacker must send successively m new Join Request messages in order to desynchronize the NS and the end-device.

Means used to achieve the attack In order to get a new Join Request message the attacker can use the technique described in Sect. 3.1 aiming at compelling the end-device to generate multiple Join Request messages. The attacker can gather several such messages and use these anytime later as "desynchronization ammunition".

Impact The consequences of this attack against the NS are the same as the one against the end-device: the targeted end-device is disconnected from the network. Unaware that the NS does not share the same security parameters, it may keep sending uplink frames for quite a long time while the NS is unable to process them. Conversely, the frames the NS may send cannot be understood by the end-device.

4 Recommendations

In this section we aim at providing recommendations that thwart the attacks described in Sect. 3. This may lead to major changes in the protocol specification and break the interoperability between patched and non-modified equipment. Hence, as an additional constraint, we aim at proposing improvements that could solve the issues as best as possible while retaining at the same time the compliance with unchanged version of end-devices or servers, in particular equipment that is already deployed and may not be easily patched.

The methods to be implemented in order to thwart the attacks against LoRaWAN must be chosen with caution. Indeed, the reduced LoRaWAN parameters size limits the efficiency of some countermeasures one may think of by paving the way to new attacks.[14] In order to preclude all the attacks, we recommend to implement all the following changes.

[14] For instance turning the `DevNonce` parameter into a counter is not a suitable solution (see Appendix B).

Generate `AppNonce` *values with no repetition* This countermeasure aims at thwarting attack A2. A counter may be used to produce the `AppNonce` values. The counter must not overlap, and one different counter must be used for each end-device in order not to artificially lower the number of activations per end-device.[15]

Detect a replay of `AppNonce` *values* This countermeasure aims at thwarting attack A1. It may be implemented using computationally and memory efficient techniques such as Bloom filters [2, 6]. However the `AppNonce` parameter being turned into a counter, it is enough for the end-device to store the last received `AppNonce` value in order to detect a replay.

Verify that the received Join Accept message corresponds to the sent Join Request message This countermeasure aims at checking that the Join Request and Join Accept messages are bound in order to thwart attack A3. We recommend to compute the `DevAddr` parameter in the following way. Let `NwkAddr` be the least 25 significant bits. `NwkAddr` is computed as $\texttt{NwkAddr} = \mathsf{H}(\texttt{DevNonce}, \texttt{AppNonce}, \texttt{DevEUI})$ where H is a collision-resistant function.

Verify that the session keys are shared This countermeasure aims at thwarting attack A4. We suggest to implement it the following way. Straight after the key exchange is done, the NS must send a so-called *DevStatusReq* command and verify (authentication tag) the *DevStatusAns* response from the end-device, or verify, if it comes earlier, the first frame sent by the end-device. The lack of response must be read into this as an issue (device or NS under attack).

In addition the NS must keep all sets of session keys from the last valid one up to the latest computed one. When the NS receives an uplink frame (carrying a *DevStatusAns* response, or another uplink frame), it checks the authentication tag with all keys, starting from the latest. If the keys that match with the authentication tag belong to one of the (currently) unapproved sets, then the NS keeps this set of session keys only and drops all the others. This set becomes then the last valid one.

5 Related work

Few analyses on LoRaWAN have been done and publicly released. Most of the public reviews deal with technical consideration such as the network management (secret keys storage, etc.) and generic attacks (e.g., hardware attacks, web attacks) unrelated to the LoRaWAN protocol. Some attacks, which exploit specific features of the protocol, are mentioned but without excess of details.

Regarding the presentation [12], no paper nor slides were made publicly available after the conference (to the best of our knowledge), however we got a summary of the talk. Yet we cannot claim to be aware of all the specifics provided during the talk.

[15] This would also make easier an attack aiming at exhausting the `AppNonce` counter (see Appendix B).

Desynchronization against an end-device According to Lifchitz [12], L'Héréec and Joulain [11], and Miller [18] a way to attack the end-device is to replay to the NS a previous Join Request message, leading to the end-device "disconnection" from the network. Conversely Zulian indicates that it is possible to replay a previous Join Accept message to an end-device ending with different session keys used by the NS and the end-device [31]. However Zulian does not exploit all the possibilities provided by such a replay. In particular, he does not envisage more devastating attacks such as our "replay or decrypt" attack.

Frame replay and frame decryption Regarding the pseudo-random `AppNonce` parameter, Lifchitz notes that it may repeat due to the birthday paradox [12]. Hence, under the strong assumption that the `DevNonce` value is "*forced*" (device controlled by an attacker), a keystream reuse is possible with high probability after $\sqrt{2\ln(2) \times 11 \times 2^{24}} \simeq 16{,}000$ activations, or 22 hours if a key exchange is done in 5 seconds. In fact, such a statement is wrong or, at least, hazy: if both `DevNonce` and `AppNonce` values repeat, this leads to a *session keys* reuse. In order to get a *keystream* reuse, it is necessary for the `DevAddr` parameter to repeat as well. Moreover this means a continuous series of key exchanges without any intermediary application frame. Hence the sake of such an attack may be questioned.

Finally this attack is unlikely successful against a NS implementing version 1.0.2 (the current 1.0 version). Indeed, according to the specification, the NS must receive a valid uplink frame protected by the new security parameters before dropping the current ones and using the new ones. The attack leads to the computation of the same session keys two different times. Yet, with high probability, these keys are fresh (i.e., never used previously by the NS with a legitimate end-device) because the attacker has no control on the `AppNonce` parameter. This means that the attacker has to forge a valid uplink frame if she wants to compel the NS to use these keys. That is the attacker must forge a valid 32-bit authentication tag (without the corresponding key). That being said, we are not aware of the LoRaWAN version analysed in that talk (1.0.1 or 1.0.2). Moreover Lifchitz does not consider the attacks doable against an end-device (without any physical intrusion on it).

Yang notes that it is possible to replay previous frames and to decrypt frames if some security parameters are reused (namely frame counter, keytsream) [28]. According to the author, this can be done if the frame counter is reset or wraps around. However the way to achieve the latter is not explained (in particular regarding an OTAA end-device). Moreover Yang's attacker targets the NS. It is unclear why the server would accept frame replays (it seems that Yang confuses the *gateway* and the NS). Similarly, the reuse of the keystream (allowing to decrypt frames) is due to a reuse of the same frame counter (with unchanged session keys). Yet, how to get the latter is not explained.

Data integrity Yang indicates that the lack of data integrity between the NS and the AS allows an attacker to modify the plaintext by changing the encrypted payload (due to the encryption in counter mode) [28].

6 Responsible disclosure

We have informed the LoRa Alliance of the vulnerabilities and the subsequent attacks against LoRaWAN 1.0. Prior to our communication, the LoRa Alliance's technical committee decided to start the development of a new version (namely 1.1). As a result of our disclosure, some countermeasures we propose have been included in the version 1.1 (turning the `AppNonce` parameter into a counter), while some features similar to other countermeasures were already included in the specification (binding the Join Request and the Join Accept messages, doing a key confirmation between the end-device and the NS).

7 Conclusion

The extensive analysis we perform of the security protocol LoRaWAN 1.0 shows that it suffers from several weaknesses. We describe precisely how these flaws can be exploited to carry out attacks, including practical ones. These attacks lead to a breach in data integrity, data confidentiality, and in the network availability.

The first type of attacks ends up with the end-device desynchronization from the network (that is the end-device is "disconnected"). The second kind allows an attacker to replay and to decrypt frames, therefore deceiving the NS (and the AS) or the end-device (which may be an actuator). The aforementioned attacks, due to the protocol flaws, do not lean on potential implementation or hardware bugs, and are likely to be successful against any equipment implementing LoRaWAN 1.0.

We present new attacks and, contrary to previous works (to the best of our knowledge), the attacks we describe target both types of equipment (end-device or NS). Moreover our attacker needs only to act on the air interface (to eavesdrop and send data), but she does not need to get a physical access to any equipment (in particular the end-device). In addition, the success of the attacks is independent from the means used to protect the secret values (e.g., using a tamper resistant module such as a Secure Element).

In addition we provide practical recommendations allowing to thwart the attacks we have found, while at the same time being compliant with the specification, and keeping the interoperability between patched and unmodified equipment. According to us, the recommended countermeasures can be implemented in a straightforward manner.

Acknowledgment

The authors thank Sébastien Canard for valuable comments and suggestions, and the anonymous reviewers for helpful comments. This article is based upon work from COST Action IC1403 CRYPTACUS, supported by COST (European Cooperation in Science and Technology).

A Duty cycle

The ducty cyle is a mechanism used to regulate the occupation rate of the radio channel by the end-device. Enforcing the duty cycle implies that an end-device cannot repeatedly send a lot of messages. Hence one could claim that the duration of the attack is greater than the figure we provide. However, the duty cycle is a regulation mechanism, not a security one (even if it could cleverly be used as such). And not all countries compel to use such a mechanism. Also, an end-device may well be certified (by the LoRa Alliance [13]) and yet not apply the duty cycle. Indeed a LoRa Alliance certification document explicitly states that "*the LoRa Certification testing will not do any duty cycle testing*" [8].

B Exhaustion attack

Generating a parameter (`DevNonce`, `AppNonce`) with no repetition, and detecting replays are some countermeasures one may think of. Yet, in LoRaWAN, size does matter. Applying one of these methods *while keeping at the same time* the original parameter size (for compliance reasons) may lead to an attack aiming at exhausting all possible `DevNonce` or `AppNonce` values, hence forbidding the NS or the end-device to start a new activation. Therefore this exhaustion attack, targeting the end-device or the NS it connects to, may lead to an irrevocable disconnection of the end-device.

B.1 Against the `DevNonce` parameter

Core If the end-device generates `DevNonce` values with no repetition or if the NS keeps track of all `DevNonce` values it receives, it is possible to disconnect the end-device once and for all.

Attack Every time the end-device sends a Join Request message, the attacker replies with a "false" Join Accept message. Hence the end-device generates a new message once again. If unique `DevNonce` values are generated, all values will be eventually used. If the NS keeps track of all `DevNonce` values, the NS will refuse further Join Request messages once all possible `DevNonce` values have been received, be these values pseudo-random or not.

Numerical example Let us assume that a key exchange is done in 5 seconds. If the `DevNonce` values never repeat, the attack targeting the end-device is achieved in $2^{16} \times 5$ seconds $= 91$ hours.

Let us consider the case when the NS keeps track of all `DevNonce` values. If the values are pseudo-random, the proportion effectively generated by the end-device, hence received by the NS after ℓ key exchanges, is $p = 1 - \exp(-\frac{\ell}{2^{16}})$. In order this proportion to be $p = 99\%$, the number of key exchanges must be at least $\ell = -2^{16} \times \ln(1-p)$. This corresponds to $\ell \simeq 301{,}804$ activations and more than 17 days to exhaust almost all `DevNonce` values. Remind that such an end-device is supposed to have an autonomous lifespan of up to ten years.

B.2 Against the AppNonce parameter

Core If the NS generates the AppNonce parameter so that it never repeats, or if the end-device keeps track of all AppNonce values it receives, then it is possible to disconnect the end-device once and for all.

Attack Let us consider the first case. The purpose is to compel the NS to use all possible AppNonce values. The NS generates a Join Accept message (hence a new AppNonce value) only if it receives a valid Join Request message. Therefore the NS must accept as many Join Request messages as possible AppNonce values. Since $|\texttt{DevNonce}| < |\texttt{AppNonce}|$, this is possible only if the NS does not keep track of all DevNonce values it receives (which is likely its behaviour). Then the attacker can use a circular list of Join Request messages. Such messages can be collected using the technique described in Sect. 3.1, and then used in a similar way as the one described in Sect. 3.1.

Note that if the NS uses the same pool of AppNonce values for all the end-devices, this leads to the definitive disconnection of all these end-devices. In such a case the attack may be distributed among several "false" end-devices (controlled by the attacker; no duty cycle enforced).

Let us consider the second case. The purpose of this attack is to make the end-device keep track, hence receive, all possible AppNonce values. This means that the NS has to accept as many Join Request messages as possible AppNonce values. Therefore the NS must not keep track of all the DevNonce values it receives (since $|\texttt{DevNonce}| < |\texttt{AppNonce}|$). Yet this is not sufficient. Indeed the end-device accepts as many Join Accept messages (hence AppNonce values) as Join Request messages it sends. Therefore if the end-device generates DevNonce values with no repetition, it limits the number of received AppNonce values. Therefore this attack is possible if the NS does not keep track of all DevNonce values, and if the end-device does not generate unique DevNonce values (which is likely their basic behaviour).

Moreover the implementation of this second case implies to be able to compel the end-device to send multiples Join Request messages *while receiving* the corresponding Join Accept responses. We have not identified such means but to be able to influence on the end-device power supply. Yet, if the end-device is switched off, it may lose memory of the stored AppNonce values, which is orthogonal to the goal of this attack.

Numerical example If the AppNonce values do not repeat, they are all produced after $2^{24} \times 5$ seconds $= 2.66$ years (using one end-device).

If the same pool of AppNonce values is used by the NS for all end-devices, the attack may be distributed among several end-devices controlled by the attacker. If 300 such end-devices are used in parallel, the attack is achieved in 3 days approximately.

If the end-device keeps track of all AppNonce values, and if the values are pseudo-random, $p = 99\%$ values are received by the end-device after $\ell = -2^{24} \times \ln(1-p) \simeq 77.26 \times 10^6$ activations. This means more than 12 years to exhaust almost all AppNonce values.

References

1. BiztechAfrica: FastNet announces Africa's first dedicated M2M network and IoT developer academy. http://www.biztechafrica.com/article/fastnet-announces-africas-first-dedicated-m2m-netw/10718/ (October 2015)
2. Bloom, B.H.: Space/Time Trade-offs in Hash Coding with Allowable Errors. Communications of the ACM 13(7), 422–426 (Jul 1970), http://doi.acm.org/10.1145/362686.362692
3. Blutetooth SIG: Bluetooth specification. https://www.bluetooth.com/specifications/adopted-specifications
4. Briodagh, K.: Japan Opens New LoRaWAN Network for IoT Testing. http://www.iotevolutionworld.com/iot/articles/423324-japan-opens-new-lorawan-network-iot-testing.htm (July 2016)
5. Diffie, W., Hellman, M.E.: Privacy and Authentication: An Introduction to Cryptography. Proceedings of the IEEE 67(3), 397–427 (Mar 1979)
6. Dillinger, P., Manolios, P.: Bloom Filters in Probabilistic Verification. In: Formal Methods in Computer-Aided Design. LNCS, vol. 3312, pp. 367–381. Springer (2004)
7. Fearn, N.: Orange deploys LoRa network in thousands of French towns. https://internetofbusiness.com/orange-lora-network-france/ (September 2016)
8. Hunt, D., Jouko, N., Ridder, M.: LoRa Alliance – End Device Certification Requirements for EU 868MHz ISM Band Devices (2016), v1.2
9. Kim, G.: Tata to deploy LoRa network for IoT. http://spectrumfutures.org/tata-to-deploy-lora-network-for-iot/ (June 2016)
10. Kinney, S.: 100 US cities covered by Senet LoRa network for IoT. http://www.rcrwireless.com/20160615/internet-of-things/100-u-s-cities-covered-senet-lora-network-iot-tag17 (June 2016)
11. L'Héréec, F., Joulain, N.: Sécurité LoRaWAN. In: Computer & Electronics Security Applications Rendez-vous – C&ESAR (2016)
12. Lifchitz, R.: Security review of LoRaWAN networks. In: Hardwear.io (2016)
13. LoRa Alliance: LoRaWAN Certified Products. https://www.lora-alliance.org/certified-products, last consulted December 8, 2017.
14. LoRa Alliance Technical committee: LoRaWAN Regional Parameters (Jul 2016), LoRa Alliance, version 1.0
15. LORIOT.io: https://www.loriot.io
16. Marek, S.: SK Telecom & KPN Deploy Nationwide LoRa IoT Networks. https://www.sdxcentral.com/articles/news/sk-telecom-kpn-deploy-nationwide-lorawan-iot-networks/2016/07/ (July 2016)
17. Mason, J., Watkins, K., Eisner, J., Stubblefield, A.: A Natural Language Approach to Automated Cryptanalysis of Two-time Pads. In: Proceedings of the 13th ACM Conference on Computer and Communications Security. pp. 235–244. CCS '06, ACM (2006), http://doi.acm.org/10.1145/1180405.1180435
18. Miller, R.: LoRa the Explorer – Attacking and Defending LoRa Systems. In: Information Security Conference – SyScan360 (2016)
19. National Institute Of Standards and Technology: NIST FIPS 197 Specification for the Advanced Encryption Standard (AES). http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf (Nov 2001)
20. National Institute Of Standards and Technology: NIST Special Publication 800-38A Recommendation for Block Cipher Modes of Operation – Methods and Techniques. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf (Dec 2001)

21. National Institute Of Standards and Technology: NIST Special Publication 800-38B Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication. http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf (May 2005)
22. SmartCitiesWorld: IoT connectivity for 100 million homes in China. https://smartcitiesworld.net/news/news/iot-connectivity-for-100-million-homes-in-china-1139 (November 2016)
23. SmartCitiesWorld: LoRaWAN IoT network deployed in Japan. https://smartcitiesworld.net/connectivity/connectivity/lorawan-iot-network-deployed-in-japan (September 2016)
24. SmartCitiesWorld: Semtech LoRa chosen for new IoT network in New Zealand. https://smartcitiesworld.net/news/news/semtech-lora-chosen-for-new-iot-network-in-new-zealand-949 (September 2016)
25. Song, J. H. and Poovendran, R. and Lee, J. and Iwata, T.: The AES-CMAC Algorithm. RFC 4493 (Jun 2006)
26. Sornin, N. and Luis, M. and Eirich, T. and Kramp, T. and Hersent, O.: LoRaWAN Specification (Jul 2016), LoRa Alliance, version 1.0.2
27. The Things Network: https://www.thethingsnetwork.org
28. Yang, X.: LoRaWAN: Vulnerability Analysis and Practical Exploitation. https://repository.tudelft.nl/islandora/object/uuid:87730790-6166-4424-9d82-8fe815733f1e/datastream/OBJ/download (2017)
29. Z-Wave Alliance: Z-Wave specification. http://z-wave.sigmadesigns.com/design-z-wave/z-wave-public-specification/
30. ZigBee Alliance: ZigBee specification. http://www.zigbee.org/download/standards-zigbee-specification/
31. Zulian, S.: Security threat analysis and countermeasures for LoRaWAN join procedure. http://tesi.cab.unipd.it/53210/ (2016)